Interval-Based Projection Method for Under-Constrained Numerical Systems

Daisuke Ishii, Alexandre Goldsztejn, Christophe Jermann

Noname manuscript No.
(will be inserted by the editor)

Interval-Based Projection Method for Under-Constrained Numerical Systems

Daisuke Ishii · Alexandre Goldsztejn · Christophe Jermann

Received: date / Accepted: date

Abstract This paper presents an interval-based method that follows the branch-and-prune scheme to compute a verified paving of a projection of the solution set of an under-constrained system. Benefits of this algorithm include anytime solving process, homogeneous verification of inner boxes, and applicability to generic problems, allowing any number of (possibly nonlinear) equality and inequality constraints. We present three key improvements of the algorithm dedicated to projection problems: (i) The verification process is enhanced in order to prove faster larger boxes in the projection space. (ii) Computational effort is saved by pruning redundant portions of the solution set that would project identically. (iii) A dedicated branching strategy allows reducing the number of treated boxes. Experimental results indicate that various applications can be modeled as projection problems and can be solved efficiently by the proposed method.

Keywords Numerical constraint programming · Interval analysis · Under-constrained systems · Projection method · Existentially quantified constraints

1 Introduction

Problems in various fields, such as control [11,12,16] and robotics [13,22], amount to characterizing a set defined by an *under-constrained* numerical constraint satisfaction problem (*NCSP*). For those under-constrained systems, which have generically an uncountable infinity of solutions, a *projection* of a solution set plays a key role in analyzing the system, especially when it becomes high dimensional. Indeed, not only resorting to projections is the only way for visualizing the solution set, but some projections can also convey a specific meaning; E.g., the projection on the pose parameters of the solutions to the kinematic equations of a robot represent

D. Ishii
National Institute of Informatics, JSPS, Tokyo, Japan
E-mail: dsksh@acm.org

A. Goldsztejn · C. Jermann
LINA, University of Nantes, CNRS, Nantes, France
E-mail: {alexandre.goldsztejn, christophe.jermann}@univ-nantes.fr

the *reachable workspace* of the considered robot, while the projection on the output parameters of the solutions of a command model represent its *response range* to the possible inputs. In many cases, a projected solution set has a positive hyper-volume[1] even though the solution set before projection has a null hyper-volume.

Example 1 Consider the variables $(x, y) \in [-2, 3] \times [-3, 1]$ and the constraint

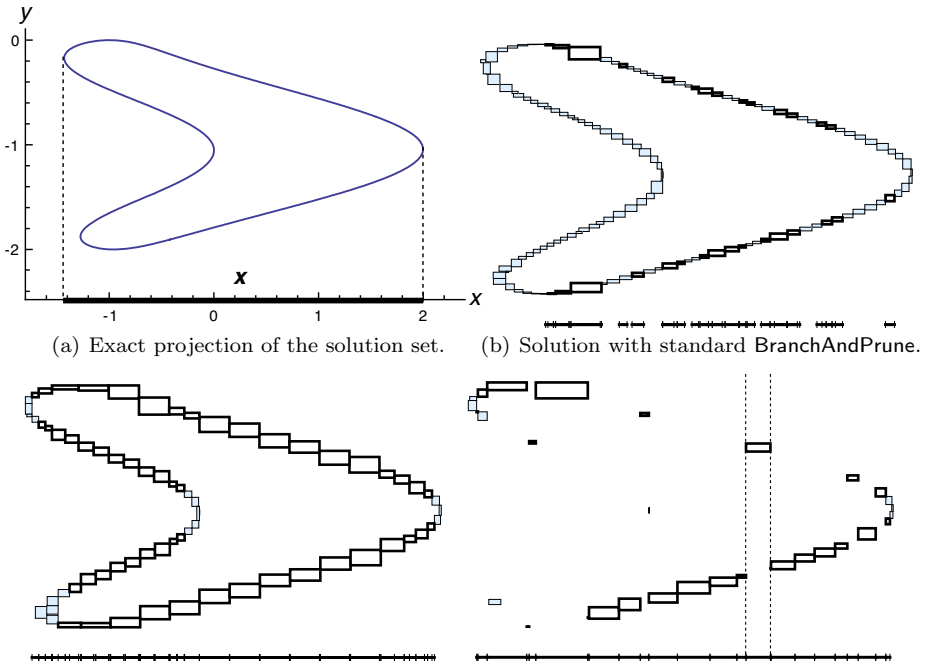
$$(x + \cos 3y)^2 + (y + 1)^2 - 1 = 0.$$

The solution set of this under-constrained NCSP is the 2D curve depicted in Figure 1(a). The projection of this solution set onto variable x is illustrated as the interval $\boldsymbol{x}$ on the x axis in the same figure.

When the constraints are polynomial, symbolic methods can solve a projection analytically (see e.g., [4]). However, symbolic methods can handle only very small systems because the size of the symbolic expressions involved grows exponentially with the number of variables and the degree of the constraints. Interval methods can handle non-polynomial constraints and they are able to compute a set of boxes (interval vectors) that approximates a projected solution set. An outer enclosure of the solution set can be computed using numerical constraint programming, but verification operators, like the interval Newton, are usually restricted to well-constrained systems. As an example, [19] is among the first work to provide an enclosure of the solutions of an under-constrained system of equations, but indeed it provides no verification. In order to be fully verified, a projected solution set has to be enclosed within two types of boxes: *Inner boxes* proved to be included inside the projected solution set; And (as small as possible) *boundary boxes* possibly containing points which are not the projection of any solution. The union of the inner and boundary boxes contains the projected solution set. The smaller the volume of the boundary boxes, the sharper the computed approximation.

The projection problem is difficult in general, and most existing interval methods were proposed to deal with specific subclasses: Linear systems [24]; Inequality systems [21]; Systems where the different constraints do not share any projected variables [11, 12]. The first method able to deal with equality constraints that share projected variables is [5]. It consists of a standard BranchAndPrune algorithm using the parametric Hansen-Sengupta operator as a verification test (i.e. Theorem 2 of this paper). This method was only applied to projection problems where projected variables domains do not need to be bisected (e.g. when the constraints depend linearly on these variables). In this case, Theorem 2 applies efficiently to verify the projection. However, this verification process becomes highly unstable as soon as the projected variables domains are bisected, which is necessary in general. This issue has already been tackled by following two different ways proposed in [6] and [7]: First, [6] proposes a specific two phase algorithm where a first phase computes an over approximation of the boundary of the projection using some necessary conditions related to critical points of projections, and uses connectivity analysis in order to identify pavings inside and outside the projection using this boundary over approximation. The second phase intends identifying part of the boundary over approximation that are in fact inside the projection by considering

[1] The hyper-volume, also called Lebesgue measure, is the generalization to arbitrary dimensions of the length in $\mathbb{R}$, the area in $\mathbb{R}^2$ and the volume in $\mathbb{R}^3$



(a) Exact projection of the solution set. (b) Solution with standard BranchAndPrune.

(c) Solution with the new verification process and search strategy. (d) Solution with the new pruning technique.

Fig. 1 Sketch of the essential components of our BranchAndPrune method on a simple projection problem.

a perturbed problem. This second phase is critical since the boundary necessary condition is not sufficient, and the points inside the projection that verify the sufficient condition (called weak boundary points in [6]) can split the projection in several fake subcomponents. Although being the first method able to verify the projection of equality constraints sharing variables whose domains require splitting, this method presents two main drawbacks: Tuning the precision of the first phase is very difficult, while the second phase is very sensitive to the output of the first phase, resulting in an algorithm quite difficult to use and tune in practice. In addition, it can only handle equality constraints, and it seems not possible to extend it to problems including inequality constraints. The second method proposed in [7] tackles the instability of verification when projected variables domains are bisected using the so-called domain inflation technique which proved to be very efficient. It is however restricted to the computation of the direct image of set defined by inequalities through a vector-valued function, which does not require a BranchAndPrune algorithm since only the projected variables are bisected, resulting in a simple bisection algorithm.

In this paper, instead of another specialized projection algorithm, we propose to use the successful paradigm of BranchAndPrune algorithms (Section 2.3) in order to tackle general projection problems (Section 3). While treating as general problems as the method in [6], this approach overcomes its disadvantages:

- BranchAndPrune algorithms are intrinsically anytime[2] as they produce arbitrarily sharp approximations of solution sets, the user being able to stop the computation when the desired accuracy of the approximation, or a resource (e.g., time) limit, is reached [2].
- When run with a sufficient precision, or enough computation time, no weak boundary box appears in this approach, which leads to a homogeneous inner approximation.
- Solving projection problems using a standard BranchAndPrune algorithm allows fully benefiting from numerical constraint programming techniques, including a native treatment of inequality constraints.

Still, the standard BranchAndPrune algorithm for under-constrained numerical constraint satisfaction problems is bound to behave poorly when used as it is for projection problems. Indeed, it will face difficulties for verifying inner boxes because it requires that the box to be verified contains a unique corresponding solution for each projected point. This will often be false due to the bisections employed during the search.

Example 2 Considering again the problem presented in Example 1, Figure 1(b) illustrates the result computed by the BranchAndPrune algorithm. The boxes in this figure form the set S, a paving of the solution set of the under-constrained NCSP. An enclosure of the projection is obtained by taking the union of all the $\boldsymbol{x}$ component of these boxes, as shown on the x axis. The thick boxes of S are those whose projection is verified to be *inner* whereas the thin boxes are not verified. The inner approximation of the projection of the solution set is the union of the $\boldsymbol{x}$ component of the verified boxes. As depicted in the figure, it is very poor in this example and consists of several disjoint parts. In fact, we can see that verified and non-verified boxes appear somehow arbitrarily in the solution set. This is typical of an inefficient verification procedure together with an unadapted search strategy.

In theory, it is impossible to devise a perfect search strategy that would bisect the search space just where needed for the verification process to succeed everywhere. Hence, in order to devise a suitable BranchAndPrune algorithm for projection problems, we propose to enhance the verification mechanism so as to compensate the wrong bisections: Our verification method presented in Section 3.1 applies an inflation technique to the considered box, dynamically shifting the parameters in order to match each of the projected points in the box. In addition, our BranchAndPrune algorithm adopts a specific search strategy which avoids unnecessary splitting and facilitates the verification process.

Example 3 Figure 1(c) presents the result of the BranchAndPrune algorithm that uses our verification method and search strategy on the problem introduced in Example 1. It shows that most of the boxes are verified thanks to our enhancements. The boxes that are not verified are too close to critical points of the projection, i.e., points where the solution set is orthogonal to the projection. No verification method could prove boxes enclosing such points.

Still, it appears much could be gained by considering the projection mechanism during the computation of the paving S. Indeed, as soon as a portion of the

[2] As long as their search strategy is *fair* at least.

projection has been verified to be *inner*, it becomes useless to pave additional parts of the solution set that would project identically. Hence the cylinder above the projection of a verified box can be used for pruning during the rest of the computation (Section 3.2).

Example 4 It is clear in Figures 1(b) and 1(c) that some redundant computations are performed because there are overlapping of up to 4 solutions within the projection. Considering verified boxes as a mean of pruning redundant portions of the search space, we can obtain the much less verbose paving depicted in Figure 1(d).

The paper is organized as follows: Section 2 introduces the necessary background on NCSPs and recalls the generic BranchAndPrune algorithm; Section 3 defines formally projection problems and details the adaptation of the BranchAndPrune algorithm to these problems, in particular the new verification process (Section 3.1), the new pruning technique (Section 3.2) and the new search strategy (Section 3.3); Section 4 presents experimental evidences of the practical usefulness and performances of our method, especially in comparison to results from the literature; Section 5 concludes the paper.

2 Numeric Constraint Solving

Numeric constraint solving inherits principles and methods from discrete constraint solving [23] and interval analysis [18]. Indeed, the specificity of the problems they address is that their variables take values in continuous subsets of $\mathbb{R}$. It is thus impossible to enumerate the possible assignments as classically done in discrete constraint solving. To overcome this situation, numeric constraint solvers resort to interval computations : Each variable takes an interval as a domain, representing the fact it can take any value within this interval. It is then possible to enumerate machine-representable interval assignments using the dichotomy principle of the BranchAndPrune scheme.

The following subsection recalls the necessary definitions of interval analysis, the following one defines numeric constraint satisfaction problems and the one after presents the classical BranchAndPrune algorithm for numeric constraint solving.

2.1 Interval Analysis

The goal of interval analysis [18,20] is to extend real computations to interval computations. It is based on the containment principle, i.e., intervals must be understood as "any possible real value in the interval" and thus the result of interval computations must be intervals enclosing any possible result of the corresponding real computations. As a tool capable of handling rigorously uncertainties and computational errors, interval analysis has become a framework of choice for verified floating-point computations on machines. Below are the basic concepts and notations from interval analysis used in this paper.

The notation of intervals in this paper conforms to the standard [14]. A (bounded) *interval* $\boldsymbol{a} = [\underline{a}, \overline{a}]$ is a connected set of real numbers $\{b \in \mathbb{R} \mid \underline{a} \leq b \leq \overline{a}\}$. $\mathbb{IR}$ denotes the set of intervals. For an interval $\boldsymbol{a}$, $\underline{a}$ and $\overline{a}$ denote the lower and upper bounds; $\operatorname{wid} \boldsymbol{a}$ denotes the width, i.e., $\overline{a} - \underline{a}$; $\operatorname{int} \boldsymbol{a}$ denotes the interior, i.e., the open interval

$\{b \in \mathbb{R} \mid \underline{a} < b < \overline{a}\}$; and $\operatorname{mid} \boldsymbol{a}$ denotes the midpoint, i.e., $(\overline{a} + \underline{a})/2$. For intervals $\boldsymbol{a}$ and $\boldsymbol{b}$, $\operatorname{dist}(\boldsymbol{a}, \boldsymbol{b})$ denotes the hypermetric between the two, i.e., $\max(|\overline{a} - \overline{b}|, |\underline{a} - \underline{b}|)$, and $\boldsymbol{a} \setminus \boldsymbol{b}$ denotes the *set difference*, i.e., $\{c \in \mathbb{R} \mid c \in \boldsymbol{a}, c \notin \operatorname{int} \boldsymbol{b}\}$, which can also be represented as the union of at most two intervals. All these definitions are naturally extended to interval vectors.

A d-dimensional *box* (or interval vector) $\boldsymbol{a}$ is a tuple of d intervals $(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_d)$. $\mathbb{IR}^d$ denotes the set of d-dimensional boxes. For a real vector $a \in \mathbb{R}^d$ and a box $\boldsymbol{a} \in \mathbb{IR}^d$, we use the inclusion notation $a \in \boldsymbol{a}$ that is interpreted as $\forall i \in \{1, \ldots, d\}$ $(a_i \in \boldsymbol{a}_i)$. A *paving* P is a set of boxes $\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k\}$ such that all $\boldsymbol{a}_i$ have the same dimension d. Boxes in a paving may overlap. We denote $\cup P$ the union of the boxes in a paving P, i.e., the set $\{a \in \mathbb{R}^d \mid \exists \boldsymbol{a}_i \in P \ (a \in \boldsymbol{a}_i)\}$.

For a function $f : \mathbb{R}^d \to \mathbb{R}$, $\boldsymbol{f} : \mathbb{IR}^d \to \mathbb{IR}$ is called an *interval extension* of f if and only if it satisfies the *containment* condition:

$$\forall \boldsymbol{a} \in \mathbb{IR}^d \ \forall a \in \boldsymbol{a} \ (f(a) \in \boldsymbol{f}(\boldsymbol{a})).$$

This definition is generalized to function vectors $F : \mathbb{R}^d \to \mathbb{R}^e$.

2.2 Numeric Constraint Satisfaction Problems

A *numeric constraint satisfaction problem* (NCSP) is defined as a triple $\mathcal{P} = \langle v, \boldsymbol{v}^{\mathrm{Init}}, c \rangle$ that consists of

- a list of *variables* $v = (v_1, \ldots, v_d)$,
- an *initial domain*, in the form of a box, represented as $\boldsymbol{v}^{\mathrm{Init}} \in \mathbb{IR}^d$, and
- a *constraint* c described as

$$F(v) = 0 \ \wedge \ G(v) \geq 0,$$

  where $F : \mathbb{R}^d \to \mathbb{R}^p$ and $G : \mathbb{R}^d \to \mathbb{R}^q$, i.e., the constraint is a conjunction of equations[3] and inequalities.

Throughout the rest of the paper, d (resp. p, q) will denote the number of variables (resp. equations, inequalities) of the considered problem. A *solution* of an NCSP is an assignment of its variables $\tilde{v} \in \boldsymbol{v}^{\mathrm{Init}}$ that satisfies its constraints. The *solution set* Σ of an NCSP is the region within $\boldsymbol{v}^{\mathrm{Init}}$ that satisfies its constraints:

$$\Sigma(\mathcal{P}) := \{v \in \boldsymbol{v}^{\mathrm{Init}} \mid c(v)\}.$$

An NCSP is said to be *over-constrained* when $d < p$ (generically, its solution set is empty), *well-constrained* when $d = p$ (generically, its solution set is discrete), and *under-constrained* when $d > p$ (generically, its solution set is continuous).

2.3 Standard BranchAndPrune for NCSPs

The BranchAndPrune algorithm is the standard complete solving method in the constraint programming paradigm. It interleaves refutation phases, known as *prune* operation, that eliminate inconsistent assignments, and exploration steps, known

Algorithm 1 BranchAndPrune algorithm.

Input: NCSP $\langle v, \boldsymbol{v}^{\text{Init}}, c\rangle$
Output: pair of lists of boxes (L, S)

```
1:  L ← {v^Init}
2:  S ← ∅
3:  while ¬Stop() do
4:     v ← Extract(L)
5:     v ← Prune(v)
6:     if v ≠ ∅ then
7:        if Prove(v) then
8:           S ← S ∪ {v}
9:        else
10:          L ← L ∪ Branch(v)
11:       end if
12:    end if
13: end while
14: return (L, S)
```

as *branch* operation, that divide the search space into parts to be processed iteratively. Algorithm 1 is a generic implementation of this scheme.

It takes as an input the problem to be solved and returns as an output a paving that distinguishes: Verified solutions in S, and unproven (or *indiscernible*) boxes in L. This implementation makes use of five abstract routines[4]:

- Stop decides when the solving is completed. In numeric constraint solving it is usually implemented as a test on the precision of the boxes in L, but it can also involve a timeout check in an anytime solving context.
- Extract selects, at each iteration, the portion of the search-space to be processed. Together with the Branch operation, it defines the search strategy of the algorithm. In numeric constraint solving, it typically implements either a depth-first search (DFS), breadth-first search (BFS) or width-first search (WFS) strategy, respectively implemented by considering L as a stack, a queue, or a priority queue ordered by the width of the contained boxes (i.e., the width of their largest intervals).
- Prune filters out inconsistent parts of the considered portion of the search-space, using various definitions of local and global consistencies. In numerical constraint solving, it typically removes only boundary portions of the considered box in order to preserve domains represented as intervals (shaving). It usually makes use of local numerical consistencies like Hull or Box consistency [17,25], or enhanced versions like MOHC [1], and global contractors like the interval Newton [7].

[3] The vectorial equation $F(v) = 0$ stands for the system $f_1(v) = 0, \ldots, f_p(v) = 0$, vectorial inequalities being defined component-wise as well.

[4] For the sake of clarity, the problem $\mathcal{P}$ and the lists L and S are supposed to be accessible from within all these routines without being explicitly passed as parameters.

- Prove verifies that the considered portion of the search-space is a solution of the problem[5]. Being a solution can take different meaning depending on the considered problem and the question asked. For instance, if the question is to find the real solution of a well-constrained NCSP, then it will generally implement a solution existence (and often uniqueness) theorem, e.g., Miranda, Brouwer or interval Newton [20], that guarantees that the considered box contains a (unique) real solution; on the other hand, if the question is to compute an inner paving of the continuous solution set of an under-constrained NCSP, then it will usually implement a solution universality test that guarantees that every real assignment in the considered box is a solution of the NCSP.
- Branch splits the considered search-space into sub-parts that are inserted back into the list L of portions to be processed. In numeric constraint solving, it typically amounts to bisecting the considered box along one of its dimension. The choice of the dimension to be split, and the split point, are elements of the search strategy of the algorithm. In general, domains are split in a round-robin mode and at their midpoints.

3 Branch-and-Prune Algorithm for Projection Problems

A *projection problem* consists in computing the projection of the solution set of an under-constrained NCSP. More precisely, a projection problem is defined as an NCSP $\mathcal{P} = \langle v, \boldsymbol{v}^{\text{Init}}, c\rangle$ and a partition of the variables $v = (x, y)$ where x and y are of size n and m respectively. The variables x and y are respectively called the projection scope and the projected variables. For commodity, the initial domain of x is denoted $\boldsymbol{x}^{\text{Init}}$ and that of y is denoted $\boldsymbol{y}^{\text{Init}}$, hence $\boldsymbol{v}^{\text{Init}} = (\boldsymbol{x}^{\text{Init}}, \boldsymbol{y}^{\text{Init}})$. The solution to a projection problem is the *projected solution set* of its NCSP and is defined as

$$\Sigma_x(\mathcal{P}) := \{x \in \boldsymbol{x}^{\text{Init}} \mid \exists y \in \boldsymbol{y}^{\text{Init}}\ (c(x, y))\},$$

which is the projection of the solution set Σ onto the x variables.

We consider projection problems where c is the conjunction of p equality constraints, where $p \leq m$, and arbitrarily many inequality constraints. This requirement on the number of equality constraints ensures that the projected solution set generically has a dimension equal to n, i.e. it has a non-null hyper-volume. In this situation, one has to compute both an inner and an outer approximation of $\Sigma_x(\mathcal{P})$, therefore the output of Algorithm 1 specialized to projection problems has the following interpretation:

$$\cup S_x \subseteq \Sigma_x(\mathcal{P}) \subseteq \cup(L_x \cup S_x), \tag{1}$$

where $S_x := \{\boldsymbol{x} \mid \exists \boldsymbol{y}\ ((\boldsymbol{x}, \boldsymbol{y}) \in S)\}$ is the projection on x of the boxes contained in S, and L_x is defined similarly from L.

In the following subsections, we show how the generic BranchAndPrune algorithm described in Algorithm 1 has to be implemented in order to compute efficiently pavings satisfying (1). In particular, the implementation of functions in Algorithm 1, i.e., Prove (Subsection 3.1), Prune (Subsection 3.2) and Branch (Subsection 3.3) dedicated to projection problems are defined below. These subsections

[5] This routine is the main difference with the discrete variant of the BranchAndPrune where it basically amounts to checking that the assignment is complete.

present techniques restricted to the well-constrained case (i.e., the number m of projected variables is equal to the number p of equations). Their adaptation to the under-constrained case (i.e., $p < m$) is presented in Subsection 3.4. Appendixes A and B provide the pseudo code that implements the subroutines of BranchAndPrune.

3.1 Verification of Inner Boxes

This section describes how we implement Prove in Algorithm 1. The goal of our implementation is to verify that the processed box is projected striclty inside Σ_x. More formally, it operates on a box $(\boldsymbol{x}, \boldsymbol{y})$ and tries to prove that $\forall x \in \boldsymbol{x}\ \exists y \in \boldsymbol{y}^{\text{Init}}\ (F(x, y) = 0)$. The verification process proposed here is based on a parametric version of the well known Hansen-Sengupta operator: Indeed, the system $F(x, y) = 0$ can be interpreted as a parametric system $F_x(y) = 0$ for which the previous quantified proposition intends proving that it has a solution for every parameters values. We first recall the verification of solutions to nonparametric systems of equations in Subsection 3.1.1. Then, we show that a parametric Hansen-Sengupta operator can be used to prove $\forall x \in \boldsymbol{x}\ \exists y \in \boldsymbol{y}\ (F(x, y) = 0)$, which is a sufficient condition for the first proposition since $\boldsymbol{y} \subseteq \boldsymbol{y}^{\text{Init}}$; we also argue that such a verification process does not lead to convergent inner approximations. Finally, we propose the *domain inflation* technique dedicated to parametric systems of equations in Subsection 3.1.3.

3.1.1 Verification of Solutions to Nonparametric Systems

The Hansen-Sengupta operator is a computationally efficient version of the interval Newton operator defined as follows: Let $F : \mathbb{R}^m \to \mathbb{R}^m$, $\boldsymbol{F}$ be an interval extension of F, $\boldsymbol{y} \in \mathbb{IR}^m$, $\tilde{y}$ be any[6] point in $\boldsymbol{y}$, $\boldsymbol{J}$ be an interval enclosure of the Jacobian of F on $\boldsymbol{y}$, and $C \in \mathbb{R}^{m \times m}$ be any[7] nonsingular matrix used as a preconditioner for $\boldsymbol{J}$,

$$\boldsymbol{H}(\boldsymbol{y}) := \tilde{y} + \boldsymbol{\Gamma}(C\boldsymbol{J}, -C\boldsymbol{F}(\tilde{y}), \boldsymbol{y} - \tilde{y}), \tag{2}$$

where

$$\boldsymbol{\Gamma}(\boldsymbol{A}, \boldsymbol{a}, \boldsymbol{b}) := (\text{Diag}^{-1}\boldsymbol{A})(\boldsymbol{a} - (\text{OffDiag}\,\boldsymbol{A})\boldsymbol{b})$$

with $\text{Diag}^{-1}\boldsymbol{A}$ the diagonal interval matrix whose diagonal entries are $(\text{Diag}^{-1}\boldsymbol{A})_{ii} = 1/a_{ii}$ and $\text{OffDiag}\,\boldsymbol{A}$ the interval matrix whose diagonal entries are null and off-diagonal entries are $(\text{OffDiag}\,\boldsymbol{A})_{ij} = a_{ij}$.

Note that this operator is defined only for systems that have exactly as many variables as equations. This requirement is mandatory for the application of interval Newton operators, and we discuss how to change under-constrained systems into well-constrained ones in Section 3.4. Note also that in this version of the operator, diagonal entries must be intervals that do not contain 0 otherwise the operator cannot be applied[8]. Finally, remark that contrarily to most forms found

[6] Usually, $\tilde{y} := \text{mid}\,\boldsymbol{y}$.

[7] The preconditioning matrix is usually chosen as $C := (\text{mid}\,\boldsymbol{J})^{-1}$ since it guarantees some good convergence properties while the approximate inversion is not too expensive for the systems of relatively small size usually handled by interval analysis.

[8] An extended version of the operator handles general interval matrices but the existence proof fails as soon as an interval that contains 0 lies on the diagonal of $\boldsymbol{J}$.

in the literature, the result of the operator presented here is not intersected with the original box $\boldsymbol{y}$. Its interpretation is summarized by the following theorem:

Theorem 1 ([10,20]) *Let $\boldsymbol{y} \in \mathbb{IR}^m$, $F : \mathbb{R}^m \to \mathbb{R}^m$ be a differentiable function. Then, every zero of F that belongs to $\boldsymbol{y}$ also belongs to $\boldsymbol{H}(\boldsymbol{y})$. Furthermore, F has a unique zero in $H(\boldsymbol{y})$ if the following condition holds:*

$$\emptyset \neq \boldsymbol{H}(\boldsymbol{y}) \subseteq \operatorname{int} \boldsymbol{y}.$$

In other words, the Hansen-Sengupta prunes a box without losing any solution, and strict inclusion of the pruned box in the initial box guarantees the existence of a unique solution.

3.1.2 Verification of Solutions to Parametric Systems

The parametric Hansen-Sengupta operator is defined as follows: Let $F(x, y) : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$, $\boldsymbol{F}$ be an interval extension of F, $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{IR}^{n+m}$, $\tilde{y}$ be any point in $\boldsymbol{y}$, $\boldsymbol{J}_y$ be an interval enclosure of the Jacobian of F on $(\boldsymbol{x}, \boldsymbol{y})$ with respect to the m projected variables y, and $C \in \mathbb{R}^{m \times m}$ be any nonsingular matrix used as a preconditioner for $\boldsymbol{J}_y$

$$\boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y}) := \tilde{y} + \boldsymbol{\Gamma}(C\boldsymbol{J}_y, -C\boldsymbol{F}(\boldsymbol{x}, \tilde{y}), \boldsymbol{y} - \tilde{y}). \tag{3}$$

Like its nonparametric version, this parametric Hansen-Sengupta operator is defined only for functions that have exactly as many components as variables y, i.e., as many equations as existentially quantified variables.

The following theorem allows using the parametric Hansen-Sengupta as a verification procedure for the projection $\boldsymbol{x}$ of a box $(\boldsymbol{x}, \boldsymbol{y})$ to be included inside Σ_x.

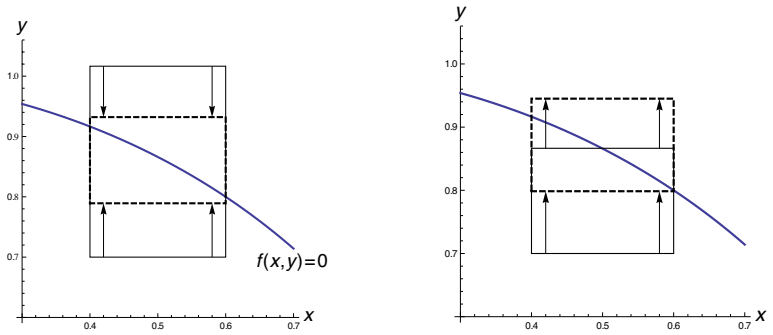
Theorem 2 ([5]) *Let $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{IR}^{n+m}$, $F(x, y) : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ be a differentiable function. Then, $\exists (x, y) \in (\boldsymbol{x}, \boldsymbol{y})$ such that $F(x, y) = 0$ implies $y \in \boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y})$. Furthermore, $\forall x \in \boldsymbol{x}\ \exists y \in \boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y})\ (F(x, y) = 0)$ holds when:*

$$\emptyset \neq \boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y}) \subseteq \operatorname{int} \boldsymbol{y}$$

Proof Using the inclusion monotonicity of the interval arithmetic, $\boldsymbol{H}_{\tilde{x}}(\boldsymbol{y}) \subseteq \boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y})$ holds for an arbitrary $\tilde{x} \in \boldsymbol{x}$. On the other hand, $\boldsymbol{H}_{\tilde{x}}(\boldsymbol{y})$ is exactly $\boldsymbol{H}(\boldsymbol{y})$ applied to the function $F(\tilde{x}, \cdot)$. Theorem 2 then directly follows from the application of Theorem 1. □

Figure 2(a) illustrates the successful application of Theorem 2: The plain box $(\boldsymbol{x}, \boldsymbol{y})$ is contracted to the dashed box $\boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y})$ (here vectors x, y and $F(x, y)$ have only one component). The parametric Hansen-Sengupta allows strictly contracting the y variable domain, and hence proves $\forall x \in \boldsymbol{x}\ \exists y \in \boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y})\ (F(x, y) = 0)$, which proves that $\boldsymbol{x} \subseteq \Sigma_x$ since $\boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y}) \subseteq \boldsymbol{y} \subseteq \boldsymbol{y}^{\mathrm{Init}}$.

Consider now the same situation but where the interval $\boldsymbol{y}$ is the half of the previous one. This is illustrated in Figure 2(b), where the failure of Theorem 2 is obvious: The quantified proposition $\forall x \in \boldsymbol{x}\ \exists y \in \boldsymbol{y}\ (F(x, y) = 0)$ is now false, and accordingly the dashed box $\boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y})$ is not anymore strictly contained in the original box. This situation is likely to happen during a bisection algorithm where



(a) Case of success. (b) Case of failure.

Fig. 2 Verification of boxes using the parametric Hansen-Sengupta.

projected variables domains have to be bisected to ensure the convergence of the computation[9].

This is dramatic for the success of the BranchAndPrune algorithm which is based on the hypothesis that the smaller the domains, the better the involved operators behave. Under this hypothesis, splitting fairly all domains usually entails the convergence of BranchAndPrune. Oppositely, we meet here a situation where a smaller domain leads to the failure of the proving process which strongly impacts the efficiency and the convergence of BranchAndPrune, as already illustrated on an example in Figure 1(b), page 3.

3.1.3 Domain Inflation Technique for Projection Problems

This key issue was addressed for a simpler class of problems in [7] by the so-called *domain inflation* technique. The idea of domain inflation can be extended to projection problems as follows: Given a box $(\boldsymbol{x}, \boldsymbol{y})$ to be processed by the Prove routine of the BranchAndPrune, instead of checking that the parametric Hansen-Sengupta is strictly contracting the domain $\boldsymbol{y}$, we use the domain $\boldsymbol{y}$ as the starting point of a search for a new domain $\boldsymbol{y}^*$ for which the parametric Hansen-Sengupta operator will be strictly contracting. Then provided that the inflated domain is included inside the initial domain, i.e., $\boldsymbol{y}^* \subseteq \boldsymbol{y}^{\text{Init}}$, the verification $\boldsymbol{x} \subseteq \Sigma_x$ succeeds. Figure 2(b) provides a hint for computing $\boldsymbol{y}^*$: Although $\boldsymbol{H_x}(\boldsymbol{y})$ is not strictly included in $\boldsymbol{y}$, it is obviously a good candidate for $\boldsymbol{y}^*$. Repeating this process leads us to consider the limit $\boldsymbol{y}^\infty$ of the sequence $\boldsymbol{y}^{k+1} := \boldsymbol{H_x}(\boldsymbol{y}^k)$, with $\boldsymbol{y}^0 := \boldsymbol{y}$. When this sequence is convergent, its limit satisfies $\boldsymbol{y}^\infty = \boldsymbol{H_x}(\boldsymbol{y}^\infty)$ and therefore defining $\boldsymbol{y}^*$ by slightly *inflating* $\boldsymbol{y}^\infty$ (i.e., increasing by a small percentage its widths) generally allows the parametric Hansen-Sengupta to be strictly contracting. For efficiency reasons, the slight inflation step is interleaved with the sequence computation so as to allow obtaining strict contraction of the parametric Hansen-Sengupta operator the soonest. The algorithm that interleaves slight inflation and the parametric

[9] Some specific classes of problems do not require splitting the projected variables y domains. In this case, the simple application of parametric existence theorems can be successful, see e.g. [5].

Hansen-Sengupta iteration is described in Appendix A where a dedicated stopping criterion allows stopping the sequence early when the parametric Hansen-Sengupta is strictly contracting or when the sequence is diverging.

This iterated parametric Hansen-Sengupta process together with the inflation technique succeeds in verifying the inner projected boxes in both Figures 2(a) and 2(b). Hence, it will typically allow verifying large portions of the projection during the BranchAndPrune process, as shown on the example in Figure 1(c), page 3.

3.2 Pruning Methods

The BranchAndPrune algorithm we propose makes use of three different pruning operations: general pruning methods implemented as the Prune routine, and more specific ones implemented within the Prove and Extract routines.

The Prune function in our algorithm can use any typical constraint propagation mechanism (e.g., *AC3*-like propagation) based on numerical contracting operators like *BC3-revise* or *HC4-revise*, that respectively enforce Box and Hull consistencies [3, 17]. This is a strength of our method to be able to make use of any pruning method that applies to standard NCSPs to address projection problems. See Algorithm 5 in Appendix B for the detail.

The goal of the Prune routine in our method is to quickly get rid of infeasible regions in order to be able to verify early large boxes. In particular, it might be counterproductive to spend a lot of time pruning a box that could be verified as is. For this reason, we simply use a cheap AC3-like propagation of HC4-revise operators in this work. Evaluating the effect of using more expensive pruning operators should be a future work since, as typically observed, stronger operators are sometimes required to address more difficult problems.

Still, it is possible to achieve strong pruning without incurring any extra cost for it by using smartly the method implemented in the Prove function. Indeed, the box $\boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y})$ computed during this routine, whether it was successfully verified or not, can ultimately be intersected with the original box $(\boldsymbol{x}, \boldsymbol{y})$ to produce a reduced box. This reduction applies only to the y variables however. The verification process we have described in Section 3.1 guarantees this reduced box contains all the solutions in the original box if any exists. Hence, by trying to verify each box in the process, we can also prune them with a strong consistency enforcing operator.

The BranchAndPrune algorithm we use computes a paving of the solution set Σ of an under-constrained NCSP, but we are only interested in its projection Σ_x. In this projection, several boxes from the computed paving may overlap, meaning several y values correspond to the same x value. This is undesirable because, to verify that any given point $\tilde{x}$ is inside the projected solution set, we need only verifying one box $(\boldsymbol{x}, \boldsymbol{y})$ where it belongs. In order to avoid unnecessary (and time-consuming) iterations of the algorithm, we thus propose to eliminate from any box to be treated all the portions whose projections correspond to already verified projections: for a box $(\boldsymbol{x}, \boldsymbol{y}) \in L$ and any verified box $(\boldsymbol{x}', \boldsymbol{y}') \in S$, we can replace $(\boldsymbol{x}, \boldsymbol{y})$ by $(\boldsymbol{x} \setminus \boldsymbol{x}', \boldsymbol{y})$ in L. This process can also be considered a pruning, except that it does not refute inconsistent portions of the search-space but avoids exploring redundant ones instead. We implement it in the Extract routine of BranchAndPrune (see Algorithm 4). A positive side-effect is that the produced projection appears more regular since no two projected boxes overlap thanks to this process.

For efficiency reasons, the fact that boxes overlap in the projected space is not recomputed each time; it is maintained all along the BranchAndPrune algorithm as a neighboring relation: two boxes $(\boldsymbol{x}, \boldsymbol{y})$ and $(\boldsymbol{x}', \boldsymbol{y}')$ in L are neighbors if their projections overlap with a non-null volume, i.e., $\text{int}\,\boldsymbol{x} \cap \boldsymbol{x}' \neq \emptyset$. Hence, the list L in BranchAndPrune stores pairs $(\boldsymbol{v}, N)$ of boxes $\boldsymbol{v}$ along with a set N of (pointers to) neighboring boxes $\boldsymbol{v}'$. Then, each time we prune (Line 5 of Algorithm 1) and split a box (Line 10 of Algorithm 1), the associated list N is updated.

This method is not only a pruning operation but also a branching one. Indeed, the result of the set difference $\boldsymbol{x} \setminus \boldsymbol{x}'$ is in general not a box, but can be represented as the union of various sets of boxes. In this case, all the resulting boxes must be pushed back into the list L. However, generating too many boxes as the result of this operation is counterproductive and can yield memory consumption problems. Still, it is possible to restrict the use of this method to reasonable cases, for instance when the result of the set difference is at most one box, i.e., the projection $\boldsymbol{x}'$ covers completely $\boldsymbol{x}$ (then $(\boldsymbol{x}, \boldsymbol{y})$ is entirely pruned) or it covers all the x dimensions but one (then the boundary of this dimension is reduced in $(\boldsymbol{x}, \boldsymbol{y})$. This is the technique we have implemented since it does not induce any additional memory issues.

Using these pruning techniques altogether allows obtaining inexpensively a regular paving of the projection as already illustrated on the example in Figure 1(d).

3.3 Search Strategy Tweaking

In the implementation of the Branch procedure of BranchAndPrune, we can tweak the search strategy in several ways in order to accommodate the projection problem (see also Algorithms 6 and 7 in Appendix B). We have chosen to adapt only the variable selection strategy and leave the split-point strategy to its default setting: balanced bisection. Indeed, there is no obvious split-point strategy that could be applied to projection problems, while there is a natural difference between selecting x or y variables.

The variable selection strategy role is to select a component of a box $\boldsymbol{v}$ to be split (because the box could not be verified yet) in order to produce sub-boxes to be pushed back into list L (Line 10 of the BranchAndPrune algorithm). Standard variable selection strategies for NCSPs are *round-robin* (RR), *largest-first* (LF) and *smear-splitting* (SS) [15]. RR selects at each application of the Branch procedure the next variable in an arbitrarily fixed order, and loops over all the variables; LF always selects the variable with the largest domain; SS selects the variable whose splitting have the highest potential impact on future prune phases according to an analysis of the partial derivatives of all constraints w.r.t. this variable.

None of them are appropriate to handle projection problems. Indeed, in order to minimize the computation time, our method aims at treating as few boxes as possible. To this end, for each point $\tilde{x}$ in the projected solution set Σ_x we want to verify a single box $(\boldsymbol{x}, \boldsymbol{y})$ such that $\tilde{x} \in \boldsymbol{x}$. For the verification process to succeed, the considered box must contain a single $\tilde{y}$ value for each possible $\tilde{x}$ value in it. Hence we need to split y variables as much as necessary to separate multiple values, but not more if possible. The existing strategies do not distinguish x and y variables and thus cannot minimize the computational effort to pave the projected solution set.

Unfortunately, no strategy can in general decide whether y variables are sufficiently split or not. Hence we propose a heuristic variable selection strategy, called the *dynamic dual round-robin* (DDRR), that relies on the other aspects of the algorithm in order to estimate whether it is still needed or not (Algorithm 7). This strategy selects in a round-robin manner all the x variables until all of them have been split s times, then it splits one y variable (also selected in a round-robin manner) and restarts considering only x variables similarly. The idea behind DDRR is to select the x variables more often than the y variables. The dynamical selection ratio s is proportional to the length of the list N of neighbor boxes, i.e., the number of boxes whose projections overlap the projection of the considered box (see Section 3.2). Indeed, the more neighbors a box has, the more it has already been split in the y dimensions and the more likely it is that the multiple y values for any given x value are already separated. We implement $s := max\{1, w \cdot length(N)\}$ where w is a predefined weight. Like RR, LF and SS, this strategy has the important property to be *fair*, i.e., all variables continue being split all along the solving process, which guarantees the convergence of the process.

3.4 Under-Constrained Projection Problems

We now explain how the routines we have defined in the previous sections can be adapted in order to address under-constrained projection problems, i.e., problems with more y variables than equations (i.e., $m > p$).

As usually done in the context of global optimization to prove the existence of a feasible point, the key idea for verifying a box is to instantiate $m - p$ projected variables to the mid-points of their domains, so that the remaining variables induce a well-constrained system. Once $m - p$ variables are instantiated, the successful application of the verification process defined in Subsection 3.1 requires that the reduced system is well-conditioned (in particular that its Jacobian is strongly regular). A Gram-Schmidt orthogonalization process applied to the mid-point of the original system allows heuristically identifying the $m-p$ variables that lead to the most well-conditioned subsystem. This heuristic ensures convergence since the size of the boxes treated during the iterations of the BranchAndPrune get smaller and smaller, and converge to zero, implying that the interval evaluation of the Jacobian gets thinner and thinner, and thus closer to its midpoint. This is confirmed by the experiments presented in Section 4.

4 Experiments

This section presents applications of the proposed method to several problems.

We have implemented our algorithm in C++. The implementation is based on the following libraries:

- The Gaol interval arithmetic library [8].
- The Realpaver library implements the generic BranchAndPrune framework [9] and uses Gaol for its computations. Each of the abstract routines Stop, Extract, Prune, Prove, and Branch in the algorithm is implemented as a class and we adapt it to our method by providing our own implementation according to the algorithms in Section 3.

Prune is implemented as an AC3-like propagation of HC4-revise operators provided by the Realpaver library. The constant weight w used within the variable selection heuristic DDRR of the search strategy is set to 0.005 based on experiments.

Section 4.1 describes the problems and provides some qualitative analysis of the results, in particular in comparison to the other methods from the state of the art. Section 4.2 presents a more detailed analysis of the performances of the proposed method, with emphasis on its asymptotic convergence. The experiments were run using a 3.4GHz Intel Xeon processor with 16GB of RAM.

4.1 Considered Problems

Three problems inspired by the existing literature were solved in our experimentation. The first example is a simple academic projection problem, and two others are more practical ones that model, respectively, the command of sailboat and a robotics problem. For each problem, we illustrate its projected solution set by presenting a paving computed by our method, using a precision check set to 10^{-2} as the Stop criterion. We also compare our paving with the paving computed by the methods in [6, 7].

4.1.1 An Academic Problem

We consider the academic problem proposed in [6]. Its solution set is defined as the intersection between a sphere and a plane. This problem is interesting because for each projected point x there exist exactly[10] two solutions (x, y_1) and (x, y_2), except on the boundary of the projection, where the two solutions merge. Proving that a projected box is inside the verified projection requires separating these two solutions, which is the main difficulty in projection problems.

We generalize this problem to a complete family of similarly defined problems $\mathrm{S\&P}_{n,m,p}$ with $n+m$ variables constrained by p equations (1 hypersphere and $p-1$ hyperplanes, $p \leq m$), projected onto n variables:

$$F(x,y) = \begin{pmatrix} x_1^2 + \cdots + x_n^2 + y_1^2 + \cdots + y_m^2 - 1 \\ x_1 + \cdots + x_n + y_1 + \cdots + y_{m-p+2} \\ x_1 + \cdots + x_n + y_2 + \cdots + y_{m-p+3} \\ \vdots \\ x_1 + \cdots + x_n + y_{p-1} + \cdots + y_m \end{pmatrix} = 0. \tag{4}$$

In [6], two instances of this problem were considered: $\mathrm{S\&P}_{2,2,2}$, a well-constrained projection problem, and $\mathrm{S\&P}_{2,3,2}$, an under-constrained projection problem. The domain for the variables is given as $x \in [-1,1]^2$ and $y \in [-0.7, 0.7] \times [-0.8, 0.8] \times [-2, 2]$.

Figure 3 and Figure 4 present the pavings of the x projections of their solution sets. They display, on the left-hand side the paving obtained with the method

[10] In fact, the under-constrained flavor of this problem has infinitely many (x, y) solutions for each projected point x, describing a full circle of solution points as the result of the intersection of a sphere and a plane. However, fixing one of the y variables to the mid-point of its domain, as explained in Section 3.4, reduces this under-constrained case to the well-constrained case, i.e., with at most two solutions for each projected point x.

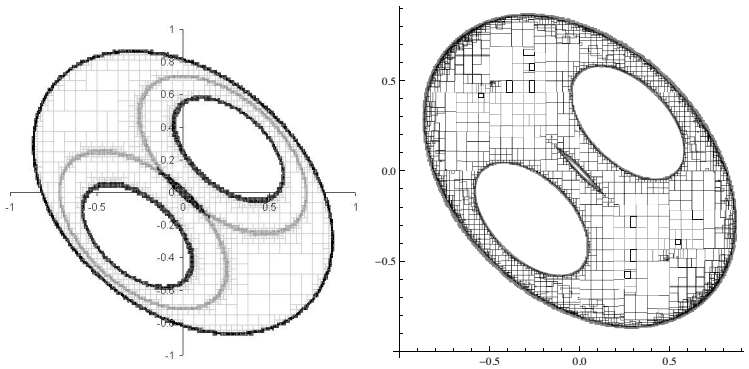


Fig. 3 Pavings for the S&P$_{2,2,2}$ problem computed by the method presented in [6] (left) and by our method (right).

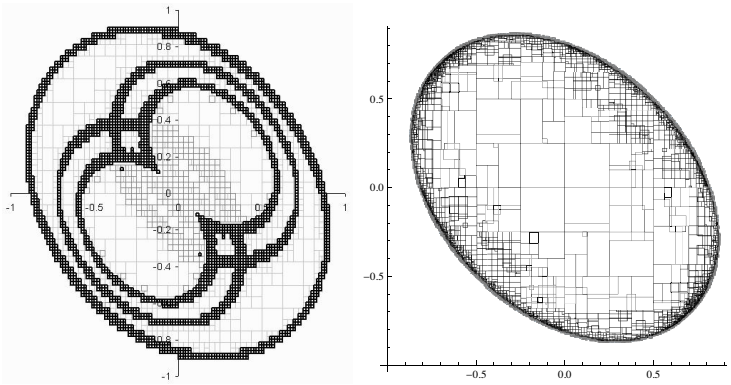


Fig. 4 Pavings for the S&P$_{2,3,2}$ problems computed by the method presented in [6] (left) and by our method (right).

described in [6] (at that time, the computation took 1 minute for S&P$_{2,2,2}$ and 15 minutes for S&P$_{2,3,2}$ respectively), and on the right-hand side the paving returned by our method (in 7.2 seconds for S&P$_{2,2,2}$ and 14 seconds for S&P$_{2,3,2}$ respectively). Interestingly, our method is not only able to reproduce the results of the previous work but also avoids to aggregate on fake boundaries located inside the pavings on the left-hand side. This is really striking on S&P$_{2,3,2}$ where the new method returns an homogeneous paving, while [6] computed a poor inner approximation with large unproved inner areas.

4.1.2 Speed Diagram of a Sailboat

The problem taken from [6,11] models the behavior of a sailboat within its control domain:

$$F(x,y) = \begin{pmatrix} \alpha_s(V\cos(x_1+y_1) - x_2\sin y_1)\sin y_1 - \alpha_r x_2 \sin^2 y_2 - \alpha_f x_2 \\ \alpha_s(V\cos(x_1+y_1) - x_2\sin y_1)(L - R_s\cos y_1) - R_r\alpha_r x_2 \sin y_2 \cos y_2 \end{pmatrix} = 0,$$

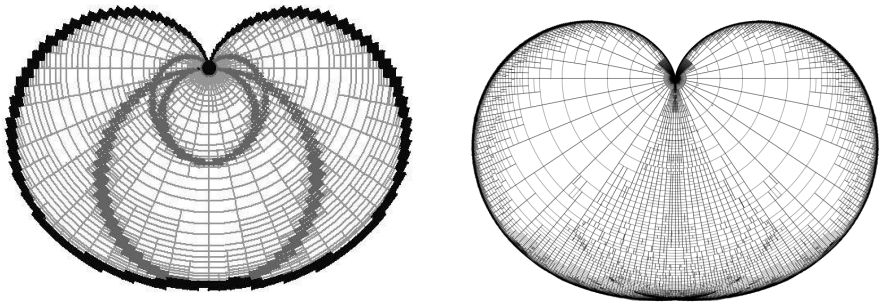


Fig. 5 Pavings of the sailboat problem computed by the method presented in [6] (left) and by our method (right).

where the variables x_1 and x_2 represent the heading angle and the speed of the sailboat, respectively, and the variables y_1 and y_2 represent the input commands for the sailboat. Given a domain for the variables, the x projection of the solution set represents the *speed diagram* of the sailboat, i.e., its response to the input commands in terms of speed and direction. The domain for each variable is given as $x \in [0, 2\pi] \times [0, 20]$ and $y \in [-\pi/2, \pi/2]^2$. The other symbols represent constants of the model of the sailboat and are defined in [11]. Here we use $\alpha_s = 100, \alpha_r = 300, \alpha_f = 60, V = 10, R_r = 2, R_s = 1$, and $L = 1$.

Figure 5 presents two pavings[11] of the speed diagram for these settings. On the left-hand side is displayed a paving obtained with the method described in [6]; At that time, it was computed in 10 minutes. On the right-hand side is displayed the paving returned by our method in 1.1 minutes. Qualitatively speaking, this figure shows again that our method is more stable than the two-phase method in [6] as it does not aggregate on fake boundaries.

4.1.3 Workspace of Serial Robots

The last problem, inspired by [7], models the workspace (reachable region) of a serial robot consisting of a bar AB (fixed or variable length) that can rotate around point A, which itself can slide along a line segment. An additional inequality constraint models collision avoidance with a circular obstacle centered at point C with radius R. Here we consider $C = (3, 1)$ and $R = 1$.

The coordinates of point B, the hand of the robot, are the x variables. The robot is commanded with two inputs: y_1 defines the position of A on the line segment (i.e., the coordinates of A are (y_1, y_1)), and y_2 is the rotation angle of the bar with respect to the horizon. The length L of the bar AB is either considered a structural parameter of the robot, i.e., it is a constant (we use $L = 2$), or it is commanded with a third input y_3 such that $L = y_3$. In the latter case, the projection becomes under-constrained. The domains of the variables are set to $x \in [-10, 10]^2$ and $y \in [0, 4] \times [-2, 2] \times [1, 2]$.

[11] Figure 5 is actually a polar diagram obtained after interpreting the computed paving in polar coordinates (x_2, x_1), x_2 being the norm and x_1 the angle.

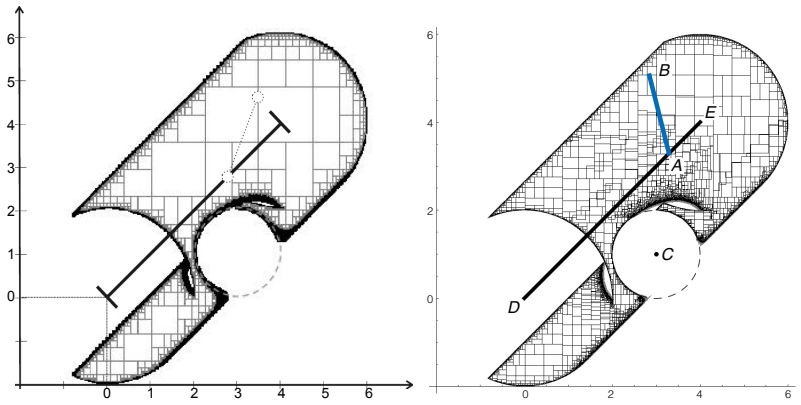


Fig. 6 Pavings of the fixed-length-bar serial robot problem computed by the method in [7] (left) and by our method (right).

The problem is then defined by

$$F(x,y) = x - \begin{pmatrix} y_1 + L\cos y_2 \\ y_1 + L\sin y_2 \end{pmatrix} = 0$$
$$\wedge\ G(x,y) = d^2(C,A,x) - R^2 \geq 0,$$

where $d^2(p,a,b)$ denotes the square of the distance separating a point p and a line segment between two points a and b, which is computed by ($\langle\cdot\,|\,\cdot\rangle$ denotes the scalar product) :

$$\begin{cases} \langle p-a\,|\,p-a\rangle & \text{if } \langle b-a\,|\,p-a\rangle < 0, \\ \langle p-b\,|\,p-b\rangle & \text{if } \langle a-b\,|\,p-b\rangle < 0, \\ \langle p-a\,|\,p-a\rangle - \dfrac{\langle b-a\,|\,p-a\rangle^2}{\langle b-a\,|\,b-a\rangle} & \text{otherwise.} \end{cases}$$

Figure 6 illustrates the pavings of the x projection (i.e., the workspace) of the fixed-length-bar robot obtained in 5.9 seconds with the precision 0.025 by the method presented in [7] (left-hand side of the figure) and by our method in 13 seconds (right-hand side of the figure). It is difficult to compare qualitatively the two methods for this problem, as they seem to produce similar results. However, note that the method in [7] is restricted to a small subclass of projection problems, for which a very efficient algorithm was proposed. Therefore, it is very positive for the new method to be competitive with the method in [7] for the problems they can both tackle.

Moreover, the method in [7] cannot handle under-constrained projection problem, and thus it cannot compute the workspace of the variable-length-bar robot. Oppositely, the new method computes this workspace in 58 seconds (see Figure 7).

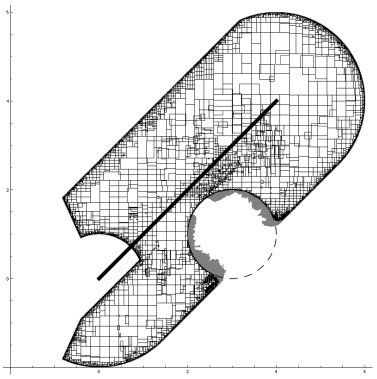


Fig. 7 Paving of the variable-length-bar serial robot problem computed with our method.

4.2 Performance Evaluation

The goal of the performance evaluation is to assess the merits of the proposed method, and more specifically the respective merits of its three main components dedicated to projection problems: The verification procedure with box inflation, the redundancy pruning by set difference, and the dedicated search strategy with dynamical dual round-robin (DDRR). To this end, we compare it with three variants, each without one of these components, and with a standard BranchAndPrune method incorporating none of our components. Hence the results we present essentially measure the impact of our implementation of the Prove, Extract and Branch routines of BranchAndPrune.

In the following, we will present three different evaluations that compare various computations in order to sort out different aspects of the practical efficiency of our method. All are based on the problems introduced in the previous section. All we use a timeout check set to 1000 seconds as the Stop criterion. Some runs may also terminate prematurely due to over-consumption of the memory, an indicator of a poor behavior of the algorithm causing too many splits and resulting in too many boxes. Finally, all are based on the same indicator, namely the *convergence speed* which measures the proportion of the volume of the projected solution set a method can verify within a certain amount of time. More formally, we measure during the experiments the proportion of the not-verified projection volume defined as

$$1 - \frac{v(t)}{\tilde{v}},$$

where $v(t)$ is the total volume of the inner projection computed at time t and $\tilde{v}$ is the exact volume of the x projection of the solution set.[12] This indicator should converge to zero when the method is convergent, and its variation over time clearly indicates the convergence speed of the method.

[12] For the problems other than S&P, the exact volume is unknown and we used the best known approximate volume instead. This approximation has a marginal impact when comparing methods relatively to one another as this value remains constant.

4.2.1 Assessment of the three components

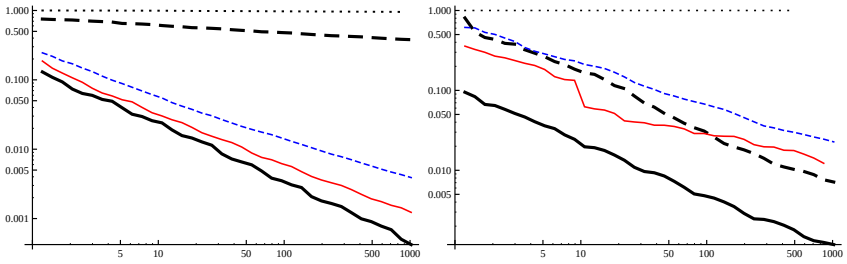
We first compare the convergence speed of the following methods on all the problems presented in Section 4.1:

A. The method with all of the components (thick plain lines).
B. The method without the box inflation method (thick dashed lines).
C. The method without the set difference method (thin plain lines).
D. The method without the DDRR strategy, i.e., standard RR is used instead (thin dashed lines).
E. The standard BranchAndPrune algorithm (dotted lines).

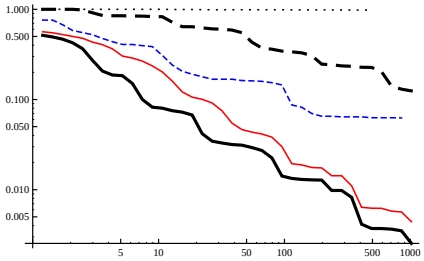
The setting A activates all of the three components we have proposed. The settings B, C and D deactivate one of the three components. The setting E applies the standard BranchAndPrune algorithm without any modification. Figure 8 illustrates the convergence speeds of the five methods with a log-log scale. A quick look at the graphs shows our method converges much faster for all tested problems. For instance, it took 30 seconds to verify 99% of the workspace of $S\&P_{2,2,2}$ for our method, compared to 49 and 180 seconds for the settings C and D, settings B and E reaching the timeout with a much lower accuracy (resp. 62% and 5%). As another evidence, after 10 seconds, our method had verified 92% of the volume of the speed diagram of the sailboat problem, against only 80%, 70%, 18% and 0% for the settings C, D, B and E, respectively. In addition, we confirmed that the results computed by our method always converged to the exact volume (or its best known approximation) of these problems.

In Figure 8(b), the graph for our method (setting A) shows sudden speed-ups of the convergence, notably around 5, 20, 80 and 200 seconds, preceded by slow-downs. Following the trace of the solving process, we note the flat portions of the curve correspond to periods of time when almost no new box is verified, hence all boxes are split iteratively until they reach a threshold allowing their verification. Because the search strategy we employ is width-first-search, the boxes in the list L are then approximately all of the same size and thus, when this threshold is reached for one of them, so it is for the others, which explains the observed speed-ups. The repetition of this behavior is explained by the relative hardness for verifying certain central regions of the speed diagram, which appears clearly in Figure 5(right) paved using a lot of similarly sized boxes. Note that the convergence speed graphs for some computations end before the timeout (1000 seconds) because the solving processes consumed all the memory of the machine (16GB).

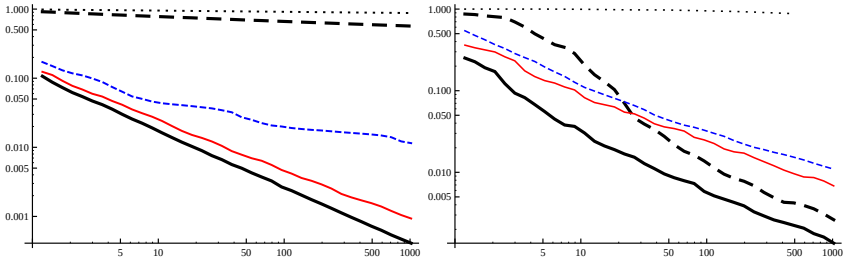
We conclude that the combination of the three components is essential in achieving quickly good verified pavings for projection problems. No component appears to be clearly dominated by the others (remark how settings B, C and D compare differently on the different problems) and none seems to have only a negligible influence in general (though setting C, without redundancy pruning by set difference, behaves quite similarly to our method on three problems in our test set, it always remain worse than setting A and behaves much less homogeneously on the whole test set). Another outcome is that, for the well-constrained projection problems, the verification with box inflation appears the most crucial as the performances without it (settings B and E) are the worst on our test set. Last but not least, this experiment illustrate why standard BranchAndPrune algorithms



(a) S&P$_{2,2,2}$ (left, $\tilde{v} \approx 1.69$); S&P$_{2,3,2}$ (right, $\tilde{v} = \pi\sqrt{3/5}$).



(b) Sailboat ($\tilde{v} \approx 25.98$).



(c) Serial robot: Fixed-length-bar robot (left, $\tilde{v} \approx 22.57$); Variable-length-bar robot (right, $\tilde{v} \approx 26.11$).

Fig. 8 Comparison of convergence speeds of the methods A–E. The horizontal and vertical axes represent the time and the proportion of the not-verified volume, respectively.

have been inappropriate to handle projection problems until now, and how it can become drastically better when adapted procedures are used instead.

4.2.2 Assessment of the neighborhood management cost

Two of our components, namely the redundancy pruning by set difference and the search strategy with DDRR, require the computation and maintenance of the neighborhood relation between boxes. This is expensive and might be a bottleneck of the solving process. This evaluation investigates the trade-off between the efficiency of our method and the cost of managing the neighborhood information. Getting rid of neighborhood management requires turning off the two components depending on it. Since we know from the previous experiment that the deactivation of one component alone can already be dramatic for the performance of the method, we expect the method without both would be terribly slow. To be

fair, we thus introduce the *static dual round-robin* strategy (DRR) that fixes the selection ratio of DDRR as $s = 1$, i.e., one y variable is split only when all x variables have been split. DRR does not require the neighborhood information while still incorporating part of the principles of DDRR. We thus compare the following settings:

A. The method with all of the components (thick plain lines).
C. The method without the set difference method (thin plain lines).
D. The method without the DDRR strategy, i.e., standard RR is used instead (thin dashed lines).
F. The method using DRR strategy instead of DDRR strategy (dotted lines).
G. The method where the set difference method is deactivated, DRR strategy is activated and the neighborhood management is deactivated (thick dashed lines).

The settings A, C and D are same as in the previous evaluation. The setting F is prepared to evaluate DRR with other components activated. It can be seen as a potential improvement over setting D. The setting G is for the computation without the neighborhood management.

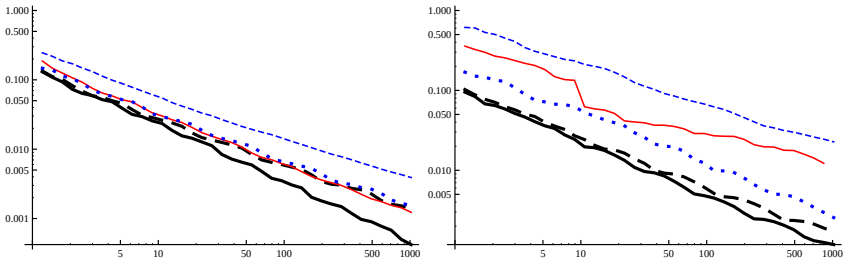
Figure 9 illustrates the convergence speeds of these settings with a log-log scale. The graphs show that the convergence speed of our method (setting A) remains always better than the other settings. The computation with the setting C is globally better than the settings F and G for the well-constrained problems. It is interesting to see that, for the under-constrained projection problems, the computation without the neighborhood management (setting G) outperforms other settings and achieves the second-best convergence speed.

This experiment consolidates the conclusions from the previous one and designates our method (setting A) with all components activated as the most efficient and robust (as it handles homogeneously well the various problems) among the tested variants.

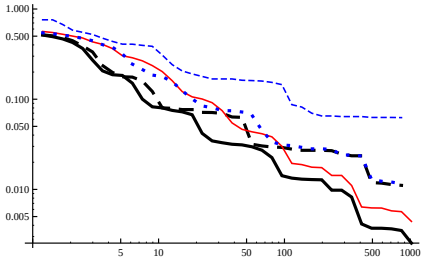
4.2.3 Effect of the Parameter Dimension

In order to evaluate the performance of our method with respect to the size of the problems, we consider a set of instances of the $\mathrm{S\&P}_{n,m,p}$ problem in which the size of the parameter variables is varied. The well-constrained instance scheme $\mathrm{S\&P}_{2,k,k}$ consists of the constraint (4), where $n = 2, m = k, p = k$, and the domain for the variables is set to $x \in [-1, 1]^2$ and $y \in [-1, 1]^k$. We solve the instances for $k = 2, 3, 4, 5, 6$ with our method (setting A from previous experiments). As the Stop criterion, we use the timeout check set to 600 seconds.

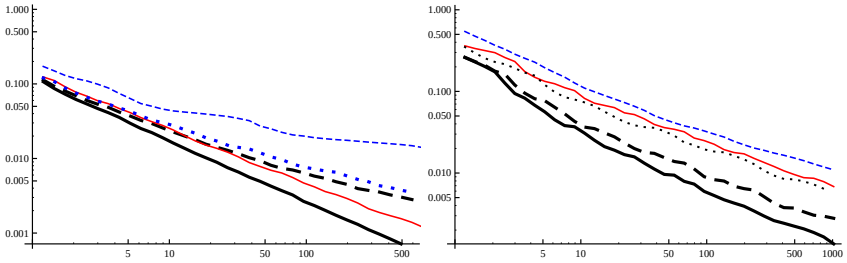
Figure 10 illustrates the convergence speed of our method with a log-log scale. The lines from bottom to top correspond to harder instances (i.e., bigger values of k), since for a fixed time less inner volume is decided as the line is higher. The exact volume $\tilde{v}$ of each instance is $\pi\sqrt{2}$, $\pi\sqrt{3/7}$, $\pi/\sqrt{3}$, $\pi\sqrt{5/17}$, and $\pi/2$, for $k = 2, 3, 4, 5, 6$, respectively. We observe that the lines in this figure are almost parallel. In a log-log scale, the slope of the line is related to the degree of the polynomial function that is displayed. Thus parallel lines mean that the complexity of the algorithm is the same for all the instances. This behavior was expected since ideally our method should compute and verify a single box for each projected point. Isolating this box requires a time that grows exponentially with the number



(a) S&P$_{2,2,2}$ (left); S&P$_{2,3,2}$ (right).



(b) Sailboat.



(c) Serial robot: Fixed-length-bar robot (left); Variable-length-bar robot (right).

Fig. 9 Comparison of convergence speeds of the methods A, C, D, F and G.

of projected variables (as for any bisection algorithm), but the convergence speed of the algorithm should remain proportional whatever this dimension, i.e., the proportion of the verified projection should grow as fast with the time for any number of projected variables. This is observed in this experiment, giving high hopes our method could deal with larger scale projection problems.

5 Conclusion

We have presented a numerical algorithm that can compute an inner and an outer approximation of the projection of the solution set to an under-constrained problem. The proposed algorithm is a simple adaptation of the standard BranchAndPrune algorithm for NCSPs. The modifications are designed as separate procedures, namely those for verification, pruning, branching strategies, and they embed heuristics for optimizing the performance of the algorithm.

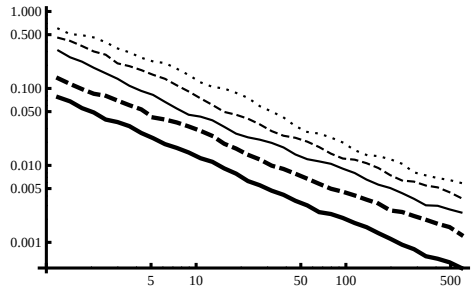


Fig. 10 Comparison of the volume computation of the S&P$_{2,k,k}$ problem ($k = 2, 3, 4, 5, 6$).

Our method is more generic than the existing interval methods in the sense that we consider systems described by an arbitrary number of variables and constraints, which can be inequality constraints. It is also more general in the sense it can make use of all the techniques available to handle efficiently constraints in the BranchAndPrune algorithm, e.g., all pruning operators are applicable. Finally, it can be adapted in order to handle under-constrained projection problems.

Many problems in control and robotics can be directly encoded as under-constrained NCSPs, and a paving of a projected solution set provides a mean for analyzing the problem. In the experiments, we have shown the applicability of our method to problems of interest in control and robotics.

The experimental results demonstrate the quality of the output of the proposed method and its efficiency with respect to prior methods, including ones much more specifically designed for the projection problem and limited in scope.

Acknowledgements This work was partially funded by the ANR grant number PSIROB06_174445 and the JSPS grant KAKENHI 23-3810. The machine used for the experiments was supported by Prof. Kazunori Ueda.

References

1. Araya, I., Trombettoni, G., Neveu, B.: Exploiting monotonicity in interval constraint propagation. In: Proc. of AAAI'10 (2010)
2. Beltran, M., Castillo, G., Kreinovich, V.: Algorithms that still produce a solution (maybe not optimal) even when interrupted: Shary's idea justified. Reliable Computing **4**(1), 39–53 (1998)
3. Benhamou, F., McAllester, D., Van Hentenryck, P.: CLP(Intervals) revisited. In: Proc. of Intl. Symp. on Logic Prog., pp. 124–138. The MIT Press (1994)
4. Collins, G.E.: Quantifier elimination by cylindrical algebraic decomposition – twenty years of progress. Quantifier Elimination and Cylindrical Algebraic Decomposition pp. 8–23 (1998)
5. Goldsztejn, A.: A Branch and Prune Algorithm for the Approximation of Non-Linear AE-Solution Sets. In: Proc. of ACM SAC 2006, pp. 1650–1654 (2006)
6. Goldsztejn, A., Jaulin, L.: Inner and outer approximations of existentially quantified equality constraints. In: Proc. of CP'06, *LNCS* 4204, pp. 198–212 (2006)
7. Goldsztejn, A., Jaulin, L.: Inner approximation of the range of vector-valued functions. Reliable Computing **14**, 1–23 (2010)
8. Goualard, F.: Gaol: NOT Just Another Interval Library (version 3.1.1). http://sourceforge.net/projects/gaol/ (2008)

9. Granvilliers, L.: Realpaver (version 1.1). http://pagesperso.lina.univ-nantes.fr/ granvilliers-l/realpaver/ (2010)
10. Hansen, E., Sengupta, S.: Bounding solutions of systems of equations using interval analysis. BIT **21**, 203–211 (1981)
11. Herrero, P., Jaulin, L., Vehi, J., Sainz, M.: Guaranteed set-point computation with application to the control of a sailboat. International journal of control, automation and systems **8**(1), 1–7 (2010)
12. Herrero, P., Sainz, M.A., Veh, J., Jaulin, L.: Quantified set inversion algorithm with applications to control. Reliable Computing **11**(5), 369–382 (2005). DOI 10.1007/s11155-005-0044-1
13. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics. Springer-Verlag, London (2001)
14. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., Van Hentenryck, P.: Standardized notation in interval analysis. In: Proc. of XIII Baikal International School-seminar "Optimization methods and their applications", pp. 106–113 (2005)
15. Kearfott, R.B., Novoa III, M.: Algorithm 681: Intbis, a portable interval newton/bisection package. ACM Trans. on Math. Softw. **16**(2), 152–157 (1990)
16. Khalil, H.K.: Nonlinear Systems, Third Edition. Prentice Hall (2002)
17. Lhomme, O.: Consistency techniques for numeric CSPs. In: Proc. of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93), pp. 232–238 (1993)
18. Moore, R.: Interval Analysis. Prentice-Hall (1966)
19. Neumaier, A.: The enclosure of solutions of parameter-dependent systems of equations. In: R. Moore (ed.) Reliability in Computing, pp. 269–286. Academic Press, San Diego (1988)
20. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press (1990)
21. Ratschan, S.: Uncertainty propagation in heterogeneous algebras for approximate quantified constraint solving. Journal of Universal Computer Science **6**(9), 861–880 (2000)
22. Reboulet, C.: Modélisation des robots parallèles. In: J.D. Boissonat, B. Faverjon, J.P. Merlet (eds.) Techniques de la robotique, architecture et commande, pp. 257–284. Hermes sciences, Paris, France (1988)
23. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA (2006)
24. Shary, S.P.: A new technique in systems analysis under interval uncertainty and ambiguity. Reliable Computing **8**(5), 321–418 (2002)
25. Van Hentenryck, P., Michel, L., Deville, Y.: Numerica : A Modeling Language for Global Optimization. MIT Press (1997)

A Verification Algorithm for Parametric Systems of Equations

Algorithm 2 interleaves the fixed point sequence $\boldsymbol{y} \leftarrow \boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y})$ (Line 3) with an inflation of $\boldsymbol{y}$ (Line 10). It returns *true* at Line 5 if and only if the Hansen-Sengupta is contracting (Line 4) and the contracted domain satisfies the inequality constraints $\boldsymbol{G}(\boldsymbol{x}, \boldsymbol{y}') \geq 0$ (Line 5). In all other cases, it returns *false*. The amounts of the last two contractions are recorded in variables d and d_{prev} in order to check the quadratic convergence of the sequence by checking that $d \leq \mu d_{prev}$ holds. This allows stopping the iteration early when the sequence does not converge. The algorithm is also stopped as soon as $\boldsymbol{y} \subseteq \boldsymbol{y}^{\mathrm{Init}}$ is not satisfied since a domain that has been shifted outside $\boldsymbol{y}^{\mathrm{Init}}$ will not come back inside this initial domain. Finite termination is enforced by requiring a maximum number of iterations in order to prevent some very untypical (but theoretically possible) cases where the sequence would converge to an exact fixed point in floating point arithmetic. Typical constants value used in our experiments are $\mu = 0.9$, $\tau = 1.01$ and $k_{max} = 10$.

Algorithm 2 Verification algorithm (Prove).

Input: NCSP $\langle (x, y), (\boldsymbol{x}, \boldsymbol{y}), c \rangle$, initial domain $\boldsymbol{y}^{\mathrm{Init}}$,
where $\boldsymbol{x} \in \mathbb{IR}^n$, $\boldsymbol{y}, \boldsymbol{y}^{\mathrm{Init}} \in \mathbb{IR}^m$, $c \equiv (F(x, y) = 0 \wedge G(x, y) \geq 0)$,
Output: $b \in \{true, false\}$

1: $d \leftarrow +\infty$; $d_{prev} \leftarrow +\infty$; $k \leftarrow 0$
2: **while** $d \leq \mu d_{prev} \wedge \boldsymbol{y} \subseteq \boldsymbol{y}^{\mathrm{Init}} \wedge k < k_{max}$ **do**
3: $\quad \boldsymbol{y}' \leftarrow \boldsymbol{H}_{\boldsymbol{x}}(\boldsymbol{y})$
4: $\quad$ **if** $\boldsymbol{y}' \subseteq \operatorname{int} \boldsymbol{y}$ **then**
5: $\quad\quad$ **return** $\boldsymbol{G}(\boldsymbol{x}, \boldsymbol{y}') \geq 0$
6: $\quad$ **end if**
7: $\quad d_{prev} \leftarrow d$
8: $\quad d \leftarrow \mathrm{dist}_{\infty}(\boldsymbol{y}, \boldsymbol{y}')$
9: $\quad k \leftarrow k + 1$
10: $\quad \boldsymbol{y} \leftarrow \operatorname{mid} \boldsymbol{y}' + \tau(\boldsymbol{y}' - \operatorname{mid} \boldsymbol{y}')$
11: **end while**
12: **return** *false*

B Implementation of BranchAndPrune

Algorithm 3 is a modified version of the standard BranchAndPrune in Algorithm 1 that implements our method. L is modified to contain pairs of a box and a list of neighbor boxes, Stop is implemented to compare the widths of boxes in L with the precision ϵ (Line 3), and additional arguments are given to each of the sub-procedures Extract, Prune, Prove and Branch. The sub-procedures are implemented as follows:

- Extract in Algorithm 4 is implemented to manage boxes with the WFS manner and to prune the extracted box by taking into account the set difference with the already verified boxes in S. Since we implement L as a list whose elements are sorted by the widths of their x components, the first element should be the widest box (Line 1). At Line 2, the algorithm computes the set difference between the extracted box and all the verified boxes. In general, a set difference results in several boxes, so we recursively compute the set difference between the resulting set of boxes and each box in S, i.e., $\boldsymbol{v} \setminus S := \{\boldsymbol{v}' \subseteq \boldsymbol{v} \mid \forall \tilde{\boldsymbol{v}} \in S \ (\boldsymbol{v}' \setminus \tilde{\boldsymbol{v}} = \emptyset)\}$. In the while loop, the algorithm updates the neighborhood information N for each box in BS (Line 5), and puts it back in L (Line 9) except that the last box in BS is returned to the main algorithm (Line 7).
- Prune in Algorithm 5 is just a wrapper of a standard filtering procedure e.g. HC4-revise. The neighborhood information should be updated after a modification to a box (Line 2).

- Prove is implemented as Algorithm 2.
- Branch in Algorithm 6 selects a variable v (Line 1), and, if there exists a variable v that can be improved (Line 2), it splits the box along the v component, otherwise it just returns the input box (Line 8). Again, the neighborhood information of the split boxes should be updated (Line 5). Algorithm 7 selects a variable of a box based on the DDRR strategy. For each box in L, the algorithm preserves the state of the previous selection using the five local counters c, i_x, i_y, n_x, n_y: i_x and i_y represent the previously selected variable from x and y, respectively; n_x and n_y counts the number of variables not selected; c is used for delaying the selection of y variables. For the first m calls, the algorithm selects a variable in x (Lines 2–7). It checks the search termination criterion before returning a variable (Line 4) and it returns the special symbol $\bullet$ if the criterion is not satisfied (Line 22). For the following call, the algorithm tries to select a variable in y (Lines 8–17). Lines 9–12 might delay the selection by resetting the counters for x variables, corresponding to the number of the neighbor boxes (w is a weight constant, which we set as $m+n$). Lines 13–17 are for selecting y variables as for x variables. When the counters i_x and i_y reach the limit, we reset the whole state and start from the first variable (Line 19).

Algorithm 3 Extended BranchAndPrune algorithm.

Input: NCSP $\langle v, \boldsymbol{v}^{\mathrm{Init}}, c\rangle$, precision ϵ
Output: pair of lists of boxes (L, S)

1: $L \leftarrow \{(\boldsymbol{v}^{\mathrm{Init}}, \emptyset)\}$
2: $S \leftarrow \emptyset$
3: **while** $\exists(\boldsymbol{v}, \cdot) \in L\ (\mathrm{wid}\,\boldsymbol{v} > \epsilon)$ **do**
4: $\quad((\boldsymbol{v}, N), L) \leftarrow \mathsf{Extract}(L, S)$
5: $\quad(\boldsymbol{v}, N) \leftarrow \mathsf{Prune}(c, \boldsymbol{v}, N)$
6: $\quad$**if** $\boldsymbol{v} \neq \emptyset$ **then**
7: $\quad\quad$**if** $\mathsf{Prove}(\langle v, \boldsymbol{v}, c\rangle, \boldsymbol{v}^{\mathrm{Init}})$ **then**
8: $\quad\quad\quad S \leftarrow S \cup \{\boldsymbol{v}\}$
9: $\quad\quad$**else**
10: $\quad\quad\quad L \leftarrow L \cup \mathsf{Branch}(\boldsymbol{v}, N)$
11: $\quad\quad$**end if**
12: $\quad$**end if**
13: **end while**
14: **return** (L, S)

Algorithm 4 Extract algorithm.

Input: sorted list of boxes L, list of verified boxes S
Output: $((\boldsymbol{v}, N), L)$

$(\boldsymbol{v}, N) \leftarrow$ take out the first element of L
$BS \leftarrow \boldsymbol{v} \setminus S$
for $\boldsymbol{v} \in BS$ **do**
  $BS \leftarrow BS \setminus \{\boldsymbol{v}\}$
  $N \leftarrow$ update N w.r.t. $\boldsymbol{v}$
  if $BS = \emptyset$ **then**
    return $((\boldsymbol{v}, N), L)$
  else
    $L \leftarrow L \cup \{(\boldsymbol{v}, N)\}$
  end if
end for

Algorithm 5 Prune algorithm.

Input: constraint c, box $\boldsymbol{v}$, list of neighbor boxes N
Output: $(\boldsymbol{v}, N)$

$\boldsymbol{v} \leftarrow \mathsf{HC4\text{-}revise}(c, \boldsymbol{v})$
$N \leftarrow$ update N w.r.t. $\boldsymbol{v}$
return $(\boldsymbol{v}, N)$

Algorithm 6 Branch algorithm.

Input: box $\boldsymbol{v}$, list of neighbor boxes N
Output: set of $(\boldsymbol{v}, N)$

$\tilde{v} \leftarrow \mathsf{DDRR}(v, \boldsymbol{v}, N)$
if $\tilde{v} \in v$ **then**
  $(\tilde{\boldsymbol{v}}, \tilde{\boldsymbol{v}}') \leftarrow$ split $\tilde{\boldsymbol{v}}$ at $\operatorname{mid} \tilde{\boldsymbol{v}}$
  $(\boldsymbol{v}, \boldsymbol{v}') \leftarrow (\boldsymbol{v}[\tilde{v} \mapsto \tilde{\boldsymbol{v}}], \boldsymbol{v}[\tilde{v} \mapsto \tilde{\boldsymbol{v}}'])$
  $(N, N') \leftarrow$ update N w.r.t. $\boldsymbol{v}$ and $\boldsymbol{v}'$, respectively
  return $\{(\boldsymbol{v}, N), (\boldsymbol{v}', N')\}$
else
  return $\{(\boldsymbol{v}, N)\}$
end if

Algorithm 7 DDRR-based variable selection algorithm.

Input: list of variables $(x_1, \ldots, x_m, y_1, \ldots, y_n)$, box $\boldsymbol{v}$, list of neighbor boxes N
(local static counters: $c \leftarrow 1$, $i_x \leftarrow 0$, $i_y \leftarrow 0$, $n_x \leftarrow 0$, $n_y \leftarrow 0$)

Output: variable v or a symbol $\bullet$

while $n_x < m$ **do**
  if $i_x < m$ **then**
    $i_x \leftarrow i_x + 1$
    if $\mathrm{wid}\, \boldsymbol{x}_{i_x} > \epsilon$ **then**
      return x_{i_x}
    end if
    $n_x \leftarrow n_x + 1$
  else if $i_y < n$ **then**
    if $c < w \cdot \mathit{length}(N)$ **then**
      $c \leftarrow c + 1$; $i_x \leftarrow 0$; $n_x \leftarrow 0$
      continue
    end if
    $i_y \leftarrow i_y + 1$
    if $\mathrm{wid}\, \boldsymbol{y}_{i_y} > \epsilon$ **then**
      $c \leftarrow 1$; $i_x \leftarrow 0$; $n_x \leftarrow 0$
      return y_{i_y}
    end if
    $n_y \leftarrow n_y + 1$
  else
    if $n_y < n$ **then**
      $i_y \leftarrow 0$; $n_y \leftarrow 0$
    else
      $i_x \leftarrow 0$; $n_x \leftarrow 0$
    end if
  end if
end while
return $\bullet$